IEEE International Conference on Communications ICC 2011
Kyoto, Japan, June 2011

Exploiting Communities for Enhancing Lookup Performance in Structured P2P Systems

H. M. N. Dilum Bandara and Anura P. Jayasumana
Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523, USA.
dilumb@engr.colostate.edu, Anura.Jayasumana@ColoState.edu

***Abstract*— Large Peer-to-Peer (P2P) systems for file transfer exhibit the presence of communities based on semantic, geographic, or organizational interests of users. Generally, resources commonly shared within individual communities are relatively unpopular and inconspicuous in the system-wide behavior. These communities are unable to benefit significantly from performance enhancement schemes such as caching that focus only on the most dominant queries. We propose a generic caching framework that enhances lookup performance of individual communities while providing even better performance to the dominant communities. The caching framework can be used with any structured P2P system that provides alternative paths to a given destination. Furthermore, the solution is adaptive to changing popularity and user interests, works with any skewed distribution of queries, needs small caches, utilizes local statistics, and introduces minimal modifications and overhead to the overlay network. Simulations based on Chord overlay show 40% reduction in average path length with individual communities indicating three times improvement in performance over system-wide caching.**

***Index Terms*—Communities, peer-to-peer, peer-to-peer caching**

I. Introduction

A. Motivation

Peer-to-Peer (P2P) systems are continuing to grow attracting millions of users and expanding into many application domains beyond conventional file sharing. Modern P2P systems share various *resources* such as files, domain names, processor cycles, storage capacity, and even sensors. Current systems are designed based on either the system-wide behavior, attempting to provide everyone an equal level of service, or optimized for more dominant users' requirements. In either case, the performance of *lookup* (i.e., the process of searching for resources) degrades as the systems continue to grow.

Recent data indicate the emergence of many small communities within a P2P system, with each community based on some common user interests [1]. A *community* is a subset of peers that share some similarity in terms of resource semantics, geography, or organizational boundaries. Members of a community with common interests may or may not be aware of each other. Peers have semantic relationships based on the type of content they frequently access [1]-[2], e.g., BitTorrent has many communities that are dedicated to music, movies, Linux distributions, games, etc. Users from the same country tend to access similar resources as well [2]-[3]. For example, for 60% of the files shared by eDoney peers, more than 80% of their replicas were located in a single country [2]. It was also observed that semantic and geographic similarities are more prominent for moderately popular files. Communities can also be formed based on organizational boundaries, e.g., peers within an autonomous system, members of a professional organization, or a group of universities often form a community to share resources and control external traffic.

Content popularity in P2P systems follow a Zipf's-like distribution [3]-[6]. However, resources popularly shared within an individual community typically do not rank high in popularity in the context of the overall P2P system [2] and often are inconspicuous in the system-wide behavior. Therefore, such communities are unable to benefit from various performance enhancements such as caching and replication that focus only on more dominant or popular resources. For example, Beehive [5] and PopCache [6] use analytical solutions to provide a guaranteed mean path length in structured P2P systems [7]. Both solutions force a large fraction of peers to cache the most popular resources regardless of their interest. In spite of requiring large caches, such solutions are inconsiderate of moderately popular resources. Several caching solutions are also available for unstructured P2P systems [8]. However, due to their random topologies, even the most popular queries are unable to gain a significant advantage from caching. Instead, several solutions propose to restructure the overlay topology based on users' interests [9]. These solutions provide better performance when a user's interests match those of the overall community. However, community membership is not rigid. A user may, for example, join multiple communities, switch from geography based community to a semantic based one, or join a geography based community in a different country. Therefore, communities do not exist in isolation, but share similarities with other communities. As shown in Table I, our analysis of search clouds from different BitTorrent communities show that user interests in most of them are somewhat similar. There are few independent communities as well, e.g., *seedpeer.com*. Three of the communities, for which we were able to estimate the popularity, show a Zipf's-like behavior with parameters

This research is supported in part by the Engineering Research Center program of the National Science Foundation under award number 0313747.

IEEE International Conference on Communications ICC 2011
Kyoto, Japan, June 2011

TABLE I – COSINE SIMILARITY AMONG DIFFERENT BITTORRENT COMMUNITIES BASED ON THEIR SEARCH CLOUDS.

Community*	EX	FE	SP	TB	TS	TE	TR
FE	0.38						
SP	0.00	0.00					
TB	0.40	0.29	0.00				
TS	0.48	0.33	0.00	0.48			
TE	0.53	0.23	0.00	0.31	0.25		
TR	0.10	0.08	0.00	0.06	0.09	0.06	
YB	0.36	0.35	0.00	0.29	0.42	0.20	0.04

* EX – extratorrent.com, FE – fenopy.com, SP – seedpeer.com, TB – torrentbit.net, TS – torrentscan.com, TE – torrentsection.com, TR – torrentreactor.net, YB – youbittorrent.com. Date – 24/07/2010 ~04:55 GMT.

0.56, 0.84, and 0.99. This further confirms that communities have different popularity distributions. Existing solutions cannot provide the same performance under such cases and are not designed to build communities based on incomparable similarity measures such as semantics and geography.

The emerging technological trends such as social networking indicate that we will continue to see the emergence of large number of small and diverse communities within large P2P systems. Future P2P architectures therefore should support such communities by providing customized services based on their distinct characteristics. Such architectures should allow the continuous formation, growth, existence, and disappearance of communities while enabling them to be a part of a large system. Conversely, the P2P system can significantly benefit by taking the characteristics and requirements of these communities into account.

B. Contributions

We propose a proactive caching framework for structured P2P systems where individual communities seamlessly cache resources that are of interest to them. Therefore, queries for popular resources within a community are resolved faster. Consequently, lookup performances of both the communities and the entire system improve. We propose a low overhead method to form sub-overlays within the overlay network and a Community Influenced Caching (CIC) algorithm. To our knowledge, this is the first caching framework for structured P2P systems that exploits communities to provide better performance. The proposed framework is independent of how communities are formed, and adaptive to changing popularity and user interests. CIC algorithm is based on local statistics, works with any skewed distribution of queries, and is computationally efficient. The algorithm reevaluates what keys to cache at the arrival of each query, and hence naturally adapts to varying query arrivals. Compared to Beehive and PoP-Cache, our framework is storage efficient and caches more distinct resources based on communities' interests. Furthermore, it introduces minimal modifications and overhead to the overlay network. Simulations based on Chord [10], for example, show 40.5% reduction in average path length with cache sizes as low as 20. Small communities were able to gain three times improvement over system-wide caching.

Section II presents the caching framework which includes the sub-overlay formation mechanism and caching algorithm. Simulation setup and performance analysis is presented in Sections III and IV, respectively. Concluding remarks and future work are presented in Section V.

II. CACHING FRAMEWORK FOR COMMUNITIES

A. Sub-overlay Formation

We focus on structured P2P systems as they are appropriate for large-scale implementations due to high scalability and some guarantees on performance [7]. These systems use a Distributed Hash Table (DHT) to index the resources. Each DHT node or a resource has a unique identifier called a *key*. Each resource or its contact information is indexed (i.e., stored) at a node having a close by key in the key space. The resources are indexed and retrieved using *put*(*key*, *value*) and *get*(*key*) messages that are forwarded to appropriate nodes using a deterministic overlay. Chord, Kademlia, CAN, and Pastry are some of the well-known solutions that are used to build such an overlay [7]. These solutions typically keep *pointers* to nodes that are spaced at exponentially increasing gaps in the key space enabling messages to be routed with a bounded path length of $O(\log N)$, where N is the number of nodes in the system.

Let us discuss a specific example using Chord [10], which is proven to be the most flexible and robust structured P2P system [11]. Chord maps both nodes and resources into a circular key space (see Fig. 1) using consistent caching. A node is assigned to a random location within the ring and a resource is indexed at its *successor*, i.e., the closest node in the clockwise direction, based on its key. Each node n maintains a set if pointers, called *fingers*, to nodes that are at $(n + 2^{i-1}) \bmod 2^m$, where $1 \le i \le m$ and m is the key length in bits. For example, node *E* in Fig. 1 keeps fingers to nodes *F*, *H*, and *J*. Routing table at a node consists of these fingers, and is called the *finger table*. Fingers are used to recursively forward a message for a given key within a bounded path length of $O(\log N)$. For example, node *E* can reach node *L* through the route $E \rightarrow J \rightarrow L$. A node may also identify redundant nodes for each of the fingers to reduce latency and enhance robustness, e.g., if *E* knows about *K*, a message may also take the path $E \rightarrow K \rightarrow L$.

Overlay messages tend to hop long distances in the key space and take alternative routes within overlay, during the first couple of hops [11]. Messages will converge in the last

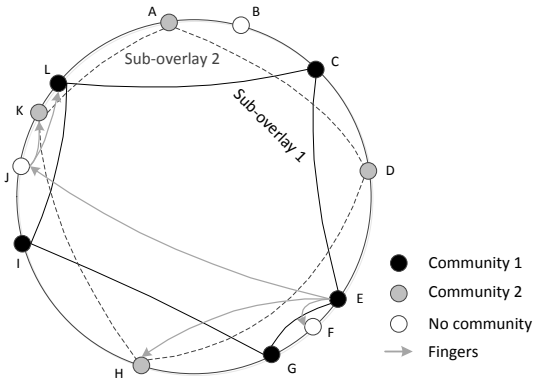

Fig. 1. Structured overlay network with a circular key space having two communities.

IEEE International Conference on Communications ICC 2011
Kyoto, Japan, June 2011

couple of hops as they reach the destination. Such a behavior provides an opportunity for a node to reach its own community members in the first few hops, and then resolve queries using their caches. For example, *E* can check *I*'s cache for a copy of a resource index in *L*. If it is not found, *I* can still forward the query to *L* within two-hops. This enables the communities to cache resources that are of interest to them while enhancing overall lookup performance. We exploit this flexibility to design a caching framework for large-scale P2P systems with multiple communities. We first propose a mechanism to identify community members and then propose an algorithm to decide what resources to cache.

Fig. 1 illustrates an overlay network having two communities. One of the communities, for example, may be based on semantics while the other may be based on geography. Some of the peers may not be in any of the communities. For the time being, assume such decisions are taken at the application layer [12], outside of the overlay or caching framework. Each of the communities has a unique *Community IDentifier* (CID). Each node indicates its communities using one or more CIDs or uses a predefined identifier to indicate that it is not in a community. Therefore, our framework supports communities based on different similarity measures or allows exceptions based on users' interest, e.g., a peer in US may connect to a community based in India just to access Hindi movies. Based on CIDs, peers try to establish stronger connections among community members allowing them to forward queries through sub-overlays as seen in Fig 1.

To build such a sub-overlay, each node needs to identify other community members that are at an exponentially increasing distance in the key space. For example, it is useful for *E* to keep pointers to *G*, *I*, and *L* instead of *F*, *H*, and *J*. To take advantage of alternative routes, we need to identify members only for the higher-order pointers/fingers, i.e., ones that point to far away nodes. To ease the identification process, each node advertises its CIDs to its successor, predecessor, and other nodes that keep pointers to it. Such advertisements can be piggybacked with overlay maintenance messages. However, given a large number of communities, it is unlikely that a node will identify members using only the advertisements that are sent to specific nodes. Nevertheless, if nodes receiving such advertisements are willing to track those CIDs within their routing tables; other nodes can query them to find community members. For example, if node *E* queries node *J*'s routing table, it can get to know about *L*. Majority of the structured P2P systems such as Chord, Pastry, and Kademlia maintain many pointers. Therefore, nodes can figure out at least one member for most of the higher-order pointers by sampling a few nodes.

We propose to use the following method to discover group members in Chord. Each node in Chord maintains at least $2\log N$ fingers and the i-th finger ($m - 2\log N \le i \le m$) covers a key space of 2^{i-1}. It can be proven that both the node pointed to by the i-th finger and its successor each has ($i + 2\log N - m$) fingers, within this range. Moreover, the successor points to ($i + 2\log N - m - 1$) distinct nodes compared to the node pointed by the i-th finger. Thus, by probing the finger table of those two nodes we can identify many nodes. If desired, the finger table of successor's successor can be probed. Moreover, a community of size M will have $M/2^{m-i+1}$ members within the coverage area of the i-th finger. Therefore, nodes are likely to find a member for large i, i.e., high-order fingers. For example, if the nodes are uniformly distributed in the key space, more members are likely to be available between pointers to nodes *J* and *E* than between nodes *F* and *H*. Chord periodically refreshes finger table by sending maintenance messages to nodes pointed by fingers. We can get those messages to probe the finger tables for community members. If a member is not found, the message is forwarded to the successor and its finger table is checked. It will be further forwarded to the successor's successor, if a member is still not found. A message should not be forwarded to the $(i + 1)^{th}$ finger node, and the maximum number of hops to forward can be defined using the parameter hop_{max}. If a member is found, its contact details can be piggybacked with the response to the maintenance message.

If a node changes its community, members of the new community can be identified by refreshing finger table either immediately or during the next cycle of overlay maintenance messages. Hence, nodes can identify relevant members with minor overhead and any structured P2P system that provides alternative routers can be used to relay messages through them. Furthermore, worse-case path length bound is still maintained as we preserve the properties of the overlay routing protocol. If routing tables have limited capacity, pointers to members can be maintained while redundant ones can be removed.

B. Community Influenced Caching Algorithm

As the messages get forwarded through community members, nodes are able to identify and cache resources that are relevant to their communities. Because we focus on communities' interest and preserve the overlay routing properties, local estimation of relative popularities is sufficient to decide what a node should cache. For example, consider a node that can cache only one resource. If messages mostly come from community members, and resource *a* is requested more frequently than resource *b*, node should cache *a*. Sometimes the node may observe even more requests for a resource *c* that is not accessed by its community. This occurs if the node is along the path to an overlay neighbor that indexes a globally popular resource (recall that alternative paths are not available as a message reaches its destination). In such a case, it is useful for the node to cache resource *c* to improve overall lookup performance. When members of the community interested in *c* realize it is a popular resource, they will add *c* to their own caches. Consequently, our node will observe a lower demand for *c*; giving it the opportunity to cache resource *a*. Therefore, in contrast to previous solutions [5]-[6],-[8], local statistics are sufficient to provide a customized service to each of the communities. Query arrivals in P2P systems show flash-crowd, diurnal, and seasonal effects [1], [5]. Therefore, statistics such as query counts or arrival rate estimates are inadequate to effectively decide when and what to cache. We propose a cach-

IEEE International Conference on Communications ICC 2011
Kyoto, Japan, June 2011

ing algorithm based on a perfect Least Frequently Used (LFC) algorithm to prevent the caches from being thrashed while ensuring overlay dynamics are sufficiently captured.

Fig. 2 illustrates the new caching algorithm, described using the common API in [12]. Each node has a cache *C*, which can stores up to C_{max} (*key*, *value*) pairs. For each query that a node receives, we track its demand using a (*key*, *demand*) pair which is stored in the lookup table *L*, where *demand* is a positive real number. Each message (*msg*) has a *source* node, message *type*, *key*, and a list of nodes (*cList*) that are interested in caching the resource (*get* messages only). ***forward( )*** is a upcall, to the DHT layer, invoked at each node that forwards a message. It enables intermediate nodes to cache, collect statistics, or drop messages. *put* messages are handled as usual. When a *get* message is received, each node keeps track of the demand for the corresponding *key* regardless of whether it is cached or not (line 4 in Fig. 2). The local cache is then checked to see whether the *msg* can be answered. If so, replies are directly sent to the source node and to the list of nodes that are interested in caching the resource (lines 5-8). The original *msg* is then dropped (line 9). If the *key* is not in the cache, the node tries to determine whether it is useful to get a copy of the resource. If the cache is already full, it checks whether the given *key* has a higher demand than the LFU cache entry (lines 12-13). A node may also request a copy, if the cache is not fully occupied and the demand is above the threshold D_{cache} (lines 17-18). In either case, the node appends, i.e., piggyback, its identifier to the *cList* in the *msg*. The *msg* is then forwarded to the next node. Intermediate nodes may also append their identifiers to the *msg*, if they also wish to cache the resource. The threshold D_{cache} reduces the caching overhead and prevents thrashing. If resources are small and relatively static (e.g., domain names) then a cache entry can be a replica of the resource. If resources are large, mutable, or have dynamic set of peers, e.g., files and processor cycles, then cache entries can point to sources of the resources. Therefore, our scheme can locate all copies of small and relatively stable resources, or subset of the copies of large, mutable, or dynamic resources. Caching algorithm works with any skewed distribution of queries and gain better performance if queries are highly skewed.

Using the ***addLookup( )*** function, nodes track the demand for keys that are not even in the cache. Thus, our caching algorithm is a perfect LFU algorithm. It enables the network to rapidly adapt to varying popularity and arrival patterns. However, it is important to properly balance past and new information to reduce the caching overhead. Therefore, whenever a new query arrives, nodes multiply demand for the key with a weighting factor α ($0 \le \alpha \le 1$). All other keys in *L* are multiplied by $(1 - \alpha)$. If α is closer to zero, we are biased towards the past thus is able to effectively respond the long-term trends. If α is closer to unity, bias is towards the current information, thus it responds to rapid changes. α and D_{cache} control the adaptability of the caching framework while minimizing unnecessary cache requests. It is recommend to set $D_{cache} > \alpha$ to prevent thrashing. If desired, communities may use different α values based on their perceived behavior. Though perfect LFU algorithms are known to take better caching decisions, they have a higher overhead as the look up table *L* can grow arbitrarily large. We use a threshold (D_{remove}) to remove keys that do not have a sufficient demand, thereby preventing *L* from growing to arbitrarily large values. Our caching algorithm is executed every time a message arrives and it is computationally efficient. Therefore, it can rapidly adapt to changing popularity, arrival patterns, and also piggyback cache requests with *get* messages while incurring minimum overhead.

```
void forward(key, msg, nextHop*)
1   If msg.type = PUT                    //put message
2       return
3   If msg.type = GET                    //get message
4     addLookup(key)                     //Track demand
5     If key ∈ C                         //In cache
6       sendDirect(msg.source, key, C[key])
7       For each i in msg.cList[ ]       //Send to each cache requester
8         sendDirect(msg.cList[i], key, C[key])
9       nextHop ← NULL                   //Drop original get message
10    Else                               //Not in cache
11      If C.size( ) = Cmax              //Cache already full
12        key_lowest ← getCachedKeyWithLowestDemand( )
13        If L[key] > L[key_lowest]      //Higher demand
14          msg.cList[ ] ← myNodeID      //Request a copy
15          remove C[key_lowest]         //Remove lowest key
16      Else
17        If L[key] > Dcache
18          msg.cList[ ] ← myNodeID      //Request a copy

void addLookup(key)
19  For each i in L[ ]
20    If i = key                         //Increase demand for key
21      L[i] = α × L[i]
22    Else                               //Decrease demand for others
23      L[i] = (1 – α) × L[i]
24    If L[i] < Dremove                  //Very low demand
25      remove L[i]
```

Fig. 2. Caching algorithm.

III. SIMULATOR

We simulated a 15,000 node network with ten communities using the OverSim P2P simulator [13]. Though hardware limitations prevented us from simulating a much larger network, size of our network is either comparable or larger than prior studies such as [5]-[6]. Community member identification mechanism and the caching algorithm were implemented on top of Chord. Nodes were assigned to different communities as shown in Table II. Parameters for each community were selected to observe behavior under different scenarios. Zipf's and similarity parameters were selected based on our own observations (Table I) and [3]-[5]. To simplify the performance analysis, we assumed a static network and queries were issued only after the network got stabilized (around 2000 s). Each node issued queries based on a Poisson distribution with a mean inter-arrival time of 15 s. Based on simulations, we observed that it is sufficient to select $D_{remove} = \alpha^{10}$ to gain good performance while limiting the lookup table size to 50-80 entries. Results are based on ten samples for which 0.95 confidence level was observed for most parameters under study.

IEEE International Conference on Communications ICC 2011
Kyoto, Japan, June 2011

IV. Performance Analysis

Fig. 3 compares the average hop count under *Chord*, *passive caching* (i.e., nodes cache responses to their queries), our *caching* algorithm without the member identification, and the overall *community caching* framework. Lookup performance converges with all caching schemes as the query distribution is steady. *Passive caching* reduces the average hop count from 7.79-hops to 7.32-hops and it is further reduced to 5.88 and 4.64 hops by *caching* and *community caching*, respectively. Thus, our caching framework reduces the average hop count of the entire network by 40.5%. Fig. 4 shows the average hop count observed by each community. Communities with highly skewed query distributions, i.e., higher Zip's parameter, lower number of distinct keys, and/or larger number of queries, gained significant performance improvement. For example, path length of communities C_3, C_7, and C_9 is reduced by 53%, 48%, and 48%, respectively. Moderately popular communities C_1, C_2, C_5, C_6, C_8, and C_{10} gained 31-42% improvement based on their specific distributions. Most popular query in C_4 had a global rank of 285 (Table II); therefore, it gained only 7.4% improvement with *caching* which focused on optimizing globally popular queries. However, *community caching* was able to reduce the hop count by 23% (3.1 times improvement over caching). Performance gain by each community was dependent only on its popularity distribution, and both large and small communities benefited equally, e.g., C_1, C_5, and C_6, and C_7 and C_9. This confirms the effectiveness of our mechanism to find community members. Fig. 5 plots the cumulative distribution of hops for each caching scheme. As expected, *community caching* was able to respond to majority of the queries within the first few hops. Chord resolved 65% of the queries within 8-hops while *community caching* was able to resolve 96% of the queries by then. With *passive caching*, some nodes respond to their own queries based on past results. This explains why it initially had a higher hit rate. We observed that cache content naturally arranges among nodes reflecting the overlay routing structure, which is explicitly defined in the design of Beehive and PoPCache. Fig. 3-5 confirm that by focusing on individual communities it is possible to improve both the communitywide and system-wide lookup performance.

Fig. 6 shows the performance gain with increasing cache size. Though average hop count rapidly reduces with increasing cache size, it tends to saturate after a while. This is an artifact of the Zipf's-like popularity distribution where significant performance can be gained by caching few highly popular resources. Yet, diminishing return is gained with very large caches. Thus, while trying to provide a guaranteed mean, both Beehive and PoPCache had to force the nodes to cache several hundred resources in average and several thousands in worse case, regardless of nodes' capabilities or interest. Yet they are able to provide only a guaranteed mean which may not be that useful compared to a guaranteed distribution. In contrast, we effectively store more distinct resources within those smaller caches because the focus is on communities' interest. It seems that PoPCache reduces the average hop count by ~38% which claims to be better than Beehive. Community C_8, with a similar popularity distribution to PoPCache simulation setup, was able to reduce the hop count by 40.5% using a cache of size 20. Therefore, our caching framework provides a best effort service with comparable performance using much smaller caches.

We observed that the network converge faster and attain minimum average path length when the weighting factor α = 0.1. Such a low value of α stabilizes the network based on long-term trends in popularity. Impact of caching threshold (D_{cache}) on convergence time is shown in Fig. 7. Higher thresholds increase the convergence time but reduce the caching overhead (see Table III). Alternatively, lower thresholds rapidly respond to popular queries consequently improving the lookup performance. This trend continues until caches gets full. When a cache is full and the threshold is lower, even a key

TABLE II – Configuration of different communities.

Community	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
No of nodes (apx.)	600	600	600	1,200	1,200	1,200	1,200	1,200	2,400	4,800
Zipf's parameter	0.85	0.95	1.10	0.5	0.80	0.80	1.0	0.90	0.90	0.75
No of distinct keys	40,000	30,000	30,000	40,000	40,000	40,000	50,000	50,000	50,000	50,000
Similarity with community (x)	0.2 (C_8)	0	0.1 (C_7)	0.2 (C_9)	0.3 (C_8) 0.5 (C_7)	0	0.1 (C_3) 0.5 (C_5)	0.3 (C_5) 0.2 (C_1)	0.4 (C_1) 0.2 (C_4) 0.3 (C_{10})	0.3 (C_9)
Queries for rank 1 key	4,516	8,535	17,100	603	6,454	6,454	21,059	11,956	23,911	17,030

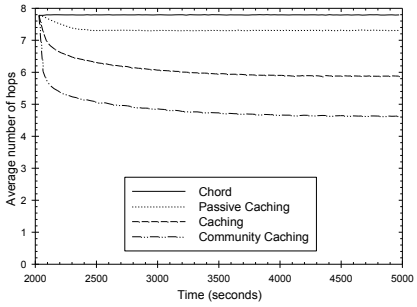

Fig. 3. Average hop count vs. time. Bucket size = 30 s, C_{max} = 20, α = 0.1, D_{cache} = 0.12, and hop_{max} = 4.

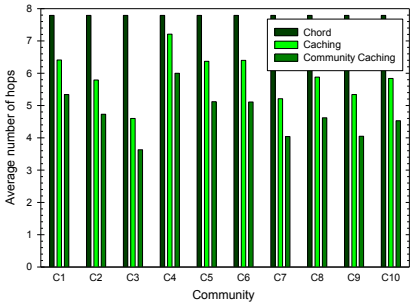

Fig. 4. Average hop count observed by each community after the steady state. C_{max} = 20, α = 0.1, D_{cache} = 0.12, and hop_{max} = 4.

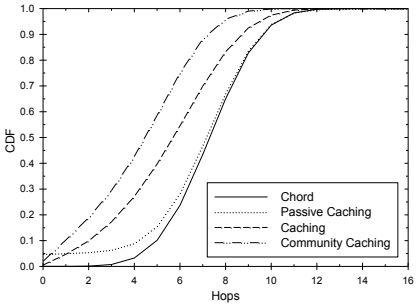

Fig. 5. Cumulative distribution of overlay hops required to resolve queries. C_{max} = 20, α = 0.1, D_{cache} = 0.12, and hop_{max} = 4.

IEEE International Conference on Communications ICC 2011
Kyoto, Japan, June 2011

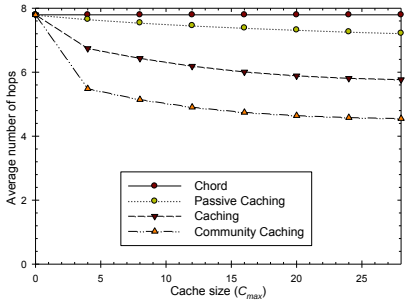

Fig. 6. Impact of maximum cache size (C_{max}) on average hop count. Steady state results with α = 0.1, D_{cache} = 0.12, and hop_{max} = 4.

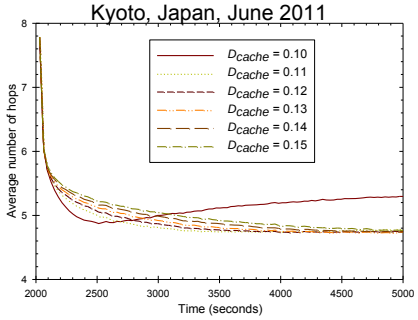

Fig. 7. Caching threshold's impact on convergence time in community caching. Bucket size = 30 s, C_{max} = 20, α = 0.1, hop_{max} = 4.

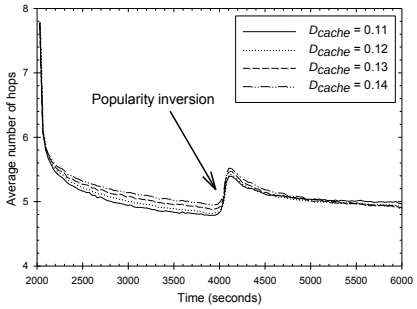

Fig. 8. Convergence of the network after popularity inversion in community caching. Bucket size = 30 s, C_{max} = 20, α = 0.1, hop_{max} = 4.

TABLE III – NUMBER OF CACHE REQUESTS IN COMMUNITY CACHING.
C_{max} = 20, α = 0.1, hop_{max} = 4

D_{cache}	0.1	0.11	0.12	0.13	0.14	0.15
Average	281.4	34.9	25.8	20.3	16.2	13.5
Minimum	0	0	0	0	0	0
Maximum	1,611.7	233.5	159.2	122.1	83.9	65.3

with a marginally higher demand than α force one of the cached keys to be flushed. When a query for the flushed key appears again, its demand increases forcing another key to be flushed. Repetition of this process results in cache thrashing and increased path length due to higher miss rate. For the given setup, α = 0.1 and D_{cache} = 0.12 balance both the convergence time and caching overhead. To observe the adaptability of our framework to rapid popularity changes, we invert the popularities of queries, where the least popular query suddenly becomes the most popular and vice versa. This is a worse-case scenario. Fig. 8 shows the convergence of the network after popularly inversion around 4000 s. Average hop count was increased by only 0.5-hops and network converges following the same convergence pattern. It was further observed that it is sufficient select hop_{max} = 4, i.e., number of hops to forward community member discovery messages. We do not expect a significant increase in hop_{max}, even for a very large network, as it is proportional to the size of a community *M*. Our framework introduces minimum overhead as caching and member discovery requests are piggybacked with query and overlay maintenance messages. Caching can alleviate hot spots within an overlay network because popular queries are answered by many nodes. Community caching framework was able to reduce the maximum number of queries answered by a Chord node from 25,151 to 1,677 (15 times reduction). Similarly, peak number of queries forwarded by a node was reduced from 27,574 to 5,191 (5.3 times reduction) by distributing the workload among community members. Thus, the proposed framework also provides good load balancing properties.

V. CONCLUSIONS AND FUTURE WORK

We propose a caching framework that allows queries to be forwarded to community members while enabling them to cache resource that of interest to their community. We present a mechanism to identify community members and a community influenced caching algorithm. The proposed framework is independent of how communities are formed, works with any skewed distribution of queries, based on local statistics, adaptive, and introduces minimum storage, network, and computational overhead. Simulation results show 23-53% improvement in communitywide and 40% improvement in system-wide lookup performance. We are currently analyzing the performance under peer churn, heterogeneous cache sizes, geography based communities, and query traces from BitTorrent communities. The concept of exploiting specific characteristics of communities can be applied to many aspects of P2P systems, and we are currently working on a solution that enables the discovery of groups of multi-attribute resources.

REFERENCES

[1] B. Zhang, A. Iosup, J. Pouwelse, D. Epema, and H. Sips, "Sampling bias in BitTorrent measurements," In Proc. *Euro-Par*, Aug.-Sep., 2010.

[2] S. B. Handurukande, A. M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin, "Peer sharing behaviour in the eDonkey network, and implications for the design of server-less file sharing systems," In Proc. *EuroSys '06*, vol. 40, no. 2, Apr. 2006, pp. 359-371.

[3] A. Klemm, C. Lindemann, M. K. Vernon, and O. P. Waldhorst, "Characterizing the query behavior in peer-to-peer file sharing systems," In Proc. *4th ACM SIGCOMM Conf. on Internet Measurement*, 2004.

[4] K. Sripanidkulchai, "The popularity of Gnutella queries and its implications on scalability," Feb. 2001, Available: http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html

[5] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays," In Proc. *USENIX NSDI '04*, vol. 1, 2004, pp. 99-112.

[6] W. Rao, L. Chen, A. W. Fu, and Y. Bu, "Optimal proactive caching in peer-to-peer network: analysis and application," In Proc. *6th ACM Con. on Information and Knowledge Management*, Nov. 2007, pp. 663-672.

[7] E. K. Lua et al., "A survey and comparison of peer-to-peer overlay network schemes," IEEE Commun. Surv. Tutor., Mar, 2004.

[8] S. A. Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys*, vol. 36, no. 4, Dec. 2004, pp. 335-371.

[9] L. Liu, J. Xu, D. Russell, and Z. Luo, "Evolution of social models in peer-to-peer networking: towards self-organising networks," In Proc. *6th Int. Conf. on Fuzzy Systems and Knowledge Discovery*, 2009.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup service for internet applications," In Proc. *ACM SIGCOMM '01*, pp. 149-160.

[11] K. Gummadi et al., "The impact of DHT routing geometry on resilience and proximity," ACM *SIGCOMM '03*, Aug. 2003, pp. 381-394.

[12] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common API for structured peer-to-peer overlays," In Proc. *Int. Workshop on Peer-To-Peer Systems*, 2003.

[13] I. Baumgart, B. Heep, and S. Krause, "OverSim: A flexible overlay network simulation framework," In Proc. *10th IEEE Global Internet Symposium*, May 2007, pp. 79-84.